Design and Implementation of a high performance IPC using Socket API

Bachelor Thesis

by

Daniel Aeneas von Rauchhaupt

University of Potsdam
Institute for Computer Science
Operating Systems and Distributed Systems

Supervisor(s):
Prof. Dr. Bettina Schnor
Max Schrötter

Potsdam, July 14, 2024

von Rauchhaupt, Daniel Aeneas
`rauchhaupt@uni-potsdam.de`
Design and Implementation of a high performance IPC using Socket API
Bachelor Thesis, Institute for Computer Science
University of Potsdam, July 2024

Mention notable people that helped you out when writing your bachelor thesis

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Potsdam, den 14. Juli 2024

Daniel Aeneas von Rauchhaupt

Abstract

This is an abstract which briefly summarizes the key points of the bachelor thesis.

Deutsche Zusammenfassung

Dies ist eine Zusammenfassung welche die Schlüsselpunkte der Bachelorarbeit kurz beschreibt.

Contents

1 Introduction

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Erat nam at lectus urna. Vitae purus faucibus ornare suspendisse sed nisi lacus. Turpis egestas integer eget aliquet nibh praesent tristique magna. Et netus et malesuada fames ac turpis egestas. Nunc vel risus commodo viverra maecenas accumsan lacus. Nisi scelerisque eu ultrices vitae auctor eu augue. Odio morbi quis commodo odio aenean sed adipiscing. Ultricies lacus sed turpis tincidunt id aliquet. Sit amet mattis vulputate enim nulla aliquet porttitor lacus luctus. Tellus rutrum tellus pellentesque eu tincidunt tortor.

Felis eget velit aliquet sagittis id consectetur purus ut faucibus. Dolor magna eget est lorem ipsum dolor sit. Sagittis aliquam malesuada bibendum arcu vitae elementum curabitur. Id interdum velit laoreet id donec ultrices tincidunt arcu. Ipsum nunc aliquet bibendum enim facilisis. Posuere morbi leo urna molestie at elementum eu facilisis. Convallis convallis tellus id interdum velit laoreet id donec ultrices. Platea dictumst quisque sagittis purus sit. Nec ultrices dui sapien eget mi proin sed libero enim. Amet justo donec enim diam vulputate ut pharetra sit amet. Sed blandit libero volutpat sed cras ornare arcu dui. Congue nisi vitae suscipit tellus mauris a diam maecenas. Ac felis donec et odio.

Eget lorem dolor sed viverra ipsum nunc aliquet. Diam volutpat commodo sed egestas egestas fringilla phasellus. Enim nunc faucibus a pellentesque sit amet. Blandit cursus risus at ultrices mi tempus imperdiet. Ornare arcu dui vivamus arcu felis bibendum ut tristique. Accumsan in nisl nisi scelerisque eu ultrices. Et tortor at risus viverra adipiscing. Convallis convallis tellus id interdum. Blandit massa enim nec dui nunc. Vitae tortor condimentum lacinia quis vel eros donec ac odio.

Justo eget magna fermentum iaculis eu. Tristique et egestas quis ipsum suspendisse ultrices gravida dictum. Luctus accumsan tortor posuere ac ut consequat semper viverra. Duis ut diam quam nulla porttitor massa id neque aliquam. Tellus mauris a diam maecenas sed enim. Et leo duis ut diam quam nulla porttitor massa. Id aliquet lectus proin nibh nisl condimentum id venenatis a. Eros in cursus turpis massa tincidunt. Sed pulvinar proin gravida hendrerit lectus. Gravida arcu ac tortor dignissim. Est ullamcorper eget nulla facilisi etiam. Netus et malesuada fames ac.

Facilisis leo vel fringilla est ullamcorper. Ultrices dui sapien eget mi. Non odio euismod lacinia at quis risus sed vulputate. Id ornare arcu odio ut. Enim ut sem viverra aliquet eget sit amet tellus cras. Sagittis orci a scelerisque purus semper eget. Nibh praesent tristique magna sit amet purus gravida quis. Adipiscing elit duis tristique sollicitudin nibh. Ultrices gravida dictum fusce ut placerat orci nulla pellentesque. Justo nec ultrices dui sapien eget mi. Nibh tellus molestie nunc non blandit massa enim nec dui. Netus et malesuada fames ac turpis egestas sed.

2 Background & Motivation

The following section establishes a definition for Host-based intrusion detection/prevention systems and introduces the example Fail2ban. An introduction to an alternative solution and its necessity, Simplefail2ban, will also be discussed. Lastly, any external tools used extensively in this thesis will also be discussed.

2.1 Host-based intrusion detection and prevention

Intrusion detection and prevention systems are tasked with monitoring the system and ensuring that no threat is present. The restriction to only utilize data available on the host system, differentiates a host-based intrusion detection system from other forms of IDS (TODO: Abbreviation). In general, this includes collecting and analyzing data, identifying outliers and responding to any potential threats or unusual behavior to minimize any potential harm. According to James P. Andersons study "Computer security threat monitoring and surveillance"[1] a threat is any deliberate attempt to either

- access data.
- manipulate data.
- or render a system unreliable or unusable.

With the ever present risk of a system having a previously unknown vulnerability, proactive measures must be taken to prevent malicious actors exploits. Real-time intrusion detection systems are required to achieve this goal. The motivation for such a system is outlined by Dorothy E. Denning[2]:

- The majority of systems have vulnerabilities, rendering them susceptible.
- Replacing systems with known vulnerabilities is difficult. Specific features may only be present in the less-secure system.
- Developing absolutely secure systems is difficult, since the explicit absence of vulnerabilities can rarely be proven.
- Secure systems remain vulnerable to insiders misusing their privileges.

For the purposes of this paper, defending against a Denial of Service, the assumption that any system is exploitable will suffice.

Host-based intrusion detection systems generally collect data from multiple sources, freely provided by the host. Such auditing of data needs to be tamper-proof and nonbypassable. Low-level system calls, often containing such data, are preferred. The anomaly based approach allows an intrusion detection system to create profiles representing legitimate behavior of clients, users and applications. Any deviation is interpreted as an attack on the system. This retains the advantage of not explicitly defining attack patterns, creating a more robust system which can identify new threats on its own.[3]

2.1.1 Fail2ban

Fail2ban is an open-source intrusion prevention system developed in Python and runs at the user space level. In contrast to a intrusion detection system, an IPS such as Fail2ban takes deliberate measures once a threat has been identified to stop attacks on a system early. By default, Fail2ban scans a variety of commonly used log files using regular expressions (regex), also called filters, to identify threats. It is therefore able to parse and monitor log data of a variety of different applications. A client will be identified as a threat if it repeatedly fails a certain task, for example establishing a TCP connection. Such a client is then banned by modifying the system firewall to deny any incoming traffic from banned IP addresses. TODO: Cite https://github.com/fail2ban/fail2ban/wiki/How-fail2ban-works

In detail, Fail2ban creates so called jails. These jails are saved on persistent storage. Therefore, restarting Fail2ban or the machine running it will not result in a loss of current jail entries. A jail consists of a log path, a certain filter, an action and a variety of customizable parameters. The filter requires at least one regex pattern. These patterns define what behavior Fail2ban should tolerate or not. An action, commonly a command or program, is to be executed once a client has been deemed a threat. Further parameters define the time the action will be active (ban time) and how often bad behavior of a client must be identified (ban limit) in log files to issue a ban. In practice, if a client fails to adhere to what the filter of a jail defines as proper behavior, vital information of that client would be deduced by the analyzed log messages. This includes to IP address of the client. A ban will then be issued and a certain action, for example dropping all traffic with the source IP of the banned client, would be performed. To issue such a ban, temporary changes to the Linux firewall, using iptables, are performed. iptables allows user space programs, such as Fail2ban, to modify, add and remove rules for packet filtering. An incoming package has to pass each set of rules before reaching the destined application. Fail2ban creates a separate rule for each banned client via iptables. New incoming packets are checked against all rules defined by iptables, or until they infringe at least one rule. Especially when many clients need to be banned, this is a clear deficit. Each banned client corresponds to one additional rule future traffic has to be compared to.[4].

2.1.2 Extended Berkeley Packet Filter

The extended Berkeley Packet Filter (eBPF) provides to opportunity to run user-generated code in a privileged setting, such as the kernel. Such eBPF programs are written in high-level programming languages, for example C. Compilers convert these programs to eBPF bytecode in user space. Successfully deploying the code requires an eBPF verifier to accept the program. This is done exclusively in kernel space to not risk the security of the operating system. If the eBPF program is accepted, the program will be converted to eBPF native code. There are several hooks to which an eBPF program can be attached to. Depending on the chosen hook, the eBPF program is deployed in or even before the network stack. Meaning, the eBPF program receives incoming traffic while the operating system is still processing it in kernel space.[4]

In this thesis, the XDP Driver hook is used for all eBPF programs. Simply put, the eBPF program and its user-generated code is run before the kernel has performed its usual processing steps for incoming traffic. Here, the program will receive each incoming packet and can decide to let it pass to the kernel unhindered, or drop it.

Since eBPF programs are event-driven, they only handle one packet at a time. In order to communicate with other programs or even store information, eBPF Maps are used. These maps are a key-vale store and provide persistent storage. However, the size of eBPF maps can not be changed during runtime and needs to be defined before creating them.[4]

This provides a significant advantage over the iptables approach of filtering packets. With eBPF programs it is possible to drop unwanted packets before they reach the computation heavy kernel network stack. And while eBPF programs have a variety of useful applications, for this thesis they are only used to either accept packets and pass them to the kernel or drop them to lighten the load.

2.1.3 Simplefail2ban

During research conducted by Florian Mikolajczak, it has been proven that Fail2ban performs poorly when dealing with large amounts of incoming unwanted traffic. This issue remained even after an alternative, and competitive, method of filtering incoming traffic using eBPF programs was implemented. To remedy this shortcoming, Simplefail2ban was developed by Paul Raatschen. It was suspected that Fail2ban was loosing performance by exclusively utilizing traditional file-based logging. The goal was to implement an IPS that can prohibit malicious actors from sending traffic to the host system, similarly to Fail2ban, without having to rely on file-based logging.

Simplefail2ban provides the option to use a shared memory section to receive log messages. This proved to be a faster method to transmit log messages from an application directly to Simplefail2ban. And whilst the method of acquisition of the log messages has changed drastically, the general requirements of banning a client has not changed compared to Fail2ban. The IPS still monitors incoming log messages for disallowed behavior. [1] Each violation of the rules imposed by Simplefail2ban results in the clients IP being logged in a hashtable. If the number of entries for one IP address is over the defined ban limit, that client is banned via one of the banning threads of Simplefail2ban. This ban is facilitated by adding the IP address to a list of banned clients with the current timestamp, and an eBPF map. An eBPF program developed by Florian Mikolajczak will check if incoming traffic should either be dropped or passed along to the kernel, depending on the eBPF map entries. The list of banned clients is routinely checked by the unbanning thread, removing clients whose ban time has elapsed from the hashtable, ban list and eBPF map.

For more details, please refer to the work of Paul Raatschen[5].

[1] Since Simplefail2ban is just a prototype, the distinction between allowed and disallowed behavior is based upon the payload of incoming traffic.

2.2 Inter-process communication

While a variety of methods for Inter-process communication exist, the nature of this thesis only necessitates the detailed explanation of both the shared memory and socket approach. As an addendum: Development was conducted on a linux based system which will be reflected when discussing technical details.

2.2.1 Shared memory approach by Paul Raatschen

During the development of his thesis, Paul Raatschen initially wanted to implement multiple IPC types. Shared memory, named pipes, sockets and message queues were all regarded as viable candidates. Ultimately, only the shared memory approach was implemented. It was considered most viable, because it did not require any involvement of the kernel during write or read operations. Hence, if the synchronization overhead for the communication processes could be kept to a minimum, the IPC could almost operate at the speed of normal memory access. Without any precedent on how to implement IPC based on shared memory, Paul Raatschen settled for an accumulation of independent segments. Each segment consists of a single ring buffer.[5]

Ring buffers are common array-like data structures. When saving data in a ring buffer, data is written in order into the buffer. Once the buffer is filled, the writing process loops back to the beginning of the array. Receiving data from a ring buffer works in a similar fashion. Once the end of the array is reached, the reader index is again set to the beginning of the ring buffer. Therefore, one can imagine a ring buffer as a circular array.

Overall, this results in data being read in a first-in first-out manner, with the index of the writing process preceding the index of the reading process. However, due to a multitude of reasons, the writer process might catch up to the index of the reader process. If this happens, there are two possible course of action. Either wait for the reader index to move and then write new data into the ring buffer, or overwrite the entry not yet read by the reader process. While overwriting the entry in the ring buffer looses data, the writer process is not slowed down by the reader process. In the shared memory approach the desired approach can be defined by setting the option “overwrite” to accept data loses.[5]
TODO: Add Shared Memory Ringbuffer from Paul Raatschen here. The outlined segments are defined via a global header, dictating certain shared variables. These included the number of ring buffers (here: segment count), the number of entries each ring buffer had (here: line count) and the size of each entry (here: line size). While other components exist in the global header, they all serve to synchronize writers and readers in one way or another and are not vital in understanding the general design of the shared memory mode. If the reader is interested in further, more technical, details on the matter, please refer to Paul Raatschens thesis[5].

Once the shared memory section has been established, multiple reader processes can attach one reading thread to each segment. Yet, per design, only one writing thread attaches to each segment. This one-to-one mapping ensures no further synchronization between multiple writer threads is required. Sending and receiving data can now be done by each thread individually according to the base principals of ring buffers outlined above.

2.2.2 Unix Domain Sockets

In order to explain what a unix domain socket is, one must understand regular internet sockets. On a linux system, a socket is a file descriptor referring to an endpoint for communication[6]. While a variety of socket types exist, the actual socket (or file descriptor representing a socket) does not change. Instead the way data is transmitted via a particular socket defines the socket type. The most common types of sockets are stream and datagram sockets.

Stream sockets provide a reliable-two way connection between communication partners. Not only do they guarantee that any data sent is transmitted without errors, but also preserve the order in which the data was sent. This behavior is achieved by utilizing the transmission control protocol (TCP).[7]

The foundation of TCP is the three-way handshake in which participants negotiate the parameters required for the data exchange. Error checking is performed on all messages. If data is corrupted, the recipient can and will request retransmission of the same data. A number of additional factors contribute to the complexity and depth of TPC. However, for this thesis the knowledge that TCP's reliability is achieved via cooperation of all participating partners will suffice.

In contrast to stream sockets, connectionless sockets, also called datagram sockets, are considered unreliable. Reason being, the usage of a different communication protocol: User Datagram Protocal (UDP). Using UDP, there is not guarantee that data will arrive at its destination. Consequently, the sequentiality of data is also not given, it may arrive in any order. The lack of an explicit connection between communication partners, instead using a best-effort service, results in lower latency during data exchange.[7]

When a socket is only represented via a path name on a local system, it is called a unix domain socket (also known as AF_UNIX). Unlike the previously discussed sockets, they are used for local only inter-process communication. Therefore, while they do inherit similar functionality as the internet sockets, they can shed slow communication protocols. Data is never sent beyond system boundaries and only handled by the kernel. There are three socket types in the UNIX domain[8]:

- SOCK_STREAM: Is a stream-oriented socket (comparable to stream sockets), establishing connections and keeping them open until explicitly closed by one communication partner.
- SOCK_DGRAM: Is a datagram-oriented socket (comparable to datagram sockets), preserving message boundaries. Additionally, SOCK_DGRAM is reliable and does not reorder sent data in most UNIX implementation.
- SOCK_SEQPACKET: Is a sequence-packet socket. It is connection-oriented, preserves message boundaries and retains the order in which data was sent.

In conclusion, unix domain sockets retain the flexibility provided by traditional internet sockets with a decrease in latency at the cost of being bound to the local system.

2.3 Packet generator: TRex

TRex is an open source traffic generator developed by Cisco Systems, capable of generating both stateless and stateful traffic[9].

TRex is based on the Data Plane Development Kit (DPDK), which is a framework promising to increase packet processing speeds for a limted number of CPU architecture. The increase in performance is mainly attributed to the Poll Mode Drivers (PMDs), which bypass the kernel's network stack.[10]

Providing the ability to use multiple cores to generate traffic, TRex can send up to 200Gb/sec with hardware supported by the DPDK framework. Utilizing Scapy, TRex is able to generate a customizable stream of traffic, allowing the user to modify any packet field. This feature will be used in this thesis to modify the source IP of all generated packets, to simulate attacks involving a large number of clients.[9]

In the scope of this thesis, TRex is used to generate UDP traffic only. The failure to achieve advertised traffic rates when using stateful traffic in certain scenarios was already observed by Paul Raatschen[5]. When deploying Simplefail2ban incoming traffic of banned clients is dropped by the IPS before reaching the network stack of the kernel. Therefore, no application receives any packets and consequently no reply is sent. This results in a loss in performance for TRex, as it expects an ACK packet when sending a TCP-SYN packet.

3 Design

The following chapter discusses the design of the socket IPC for Simplefail2ban. While reasoning the choice of unix domain sockets, advantages and disadvantages are presented.

3.1 Reasoning for Unix Domain Sockets

In order to make an informed decision on which IPC type is suited best for an IPC method, requirements need to be specified. Since the task at hand is to defend against DoS like attacks on the host system, the following aspects are considered[5]:

- **Low latency**
  Responding quickly to incoming threats is key to successfully block incoming attacks. A quick transfer of data to the IPS facilitates faster banning of malicious attackers, before they can overwhelm the system. In general, low overhead is required to achieve these goals.

- **High bandwidth**
  Considering that the host system is bombarded with millions of packets each second during an ongoing DoS attack, high bandwidth is crucial. To avoid bottlenecks between the writer and the IPS, large amounts of data need to be transmissible at once instead of requiring separate transmissions.

- **Reliability**
  Ensuring that no crucial log messages are lost due to unreliability is desirable. Repeatedly missing information about malicious clients delays the response time of the IPS, risking uptime of the system and its services.

- **Scalability**
  Log messages can come from a multitude of sources and contain a variety of information. Multiple applications should be able to submit log messages to the IPS at once and retain the possibility of providing it to other applications. Therefore, the option to have both multiple readers and writers needs to be present. Furthermore, while not necessary for the development of a host-based IPS, the option to scale beyond the local filesystem is interesting.

- **Portability**
  Developing an IPS requires it to actually be usable with already existing applications. Whereas this thesis is not intended to be more than a demonstration of a proof of concept, potential future development still requires some flexibility. A well defined API that can realistically be integrated into any real application without the need for specific hard- and software is a bare minimum.

Initially, the decision by Paul Raatschen to use shared memory as the IPC method used in Simplefail2ban was mainly based on the fact that the kernel was not involved in any read or write operations[5]. Yet, no other IPC method was implemented. To ensure that the shared memory approach is the most viable, an alternative needed to be chosen to be measured against.

The choice fell on unix domain sockets, due to the already existing read and write API and its support in the Posix standard on all UNIX systems since at least 2007[11]. Additionally, the utilization of sockets provided a great deal of flexibility during usage of the IPC which will be discussed in the next section.

3.2 Design and abstractions

During the design process of the socket architecture it was decided that supporting attachment of multiple readers was essential, as already discussed in section **??**. Each reader should receive all log messages sent by any writers. These restrictions led to the design illustrated in **??**. TODO: Update this graphic + Dataflow

The figure displays the general data flow using the socket IPC. Each reader application creates its own UNIX domain socket. Sockets are bound to a filesystem pathname. Readers can receive data from their own socket without having to compete or synchronize with other readers for data thanks to the one-to-one mapping between socket and reader. Meanwhile, writers can also independently write data into sockets without having to communicate with other writer processes. In order to guarantee that readers receive all data being sent via the socket IPC architecture, writers need to periodically recheck for newly opened sockets and always send their data to all available sockets. This results in the writers having a significant portion of the overhead, needing to resend identical data multiple times. Minimizing overhead on the reader side is important to maximize the limited computational time crucial services, such as the IPS Simplefail2ban, need to process incoming log messages.

To preserve the integrity of log messages, the unix domain socket needs to retain message boundaries, ruling out the SOCK_STREAM unix domain socket presented in section **??**. Without the absolute guarantee of reliable behavior in regard to reordering of messages - which is only present in most UNIX implementation but not all[8] - SOCK_DGRAM is unappealing as well. Therefore, the socket type choice falls on SOCK_SEQPACKET, a connection-oriented option that retains message-boundaries and sequence.

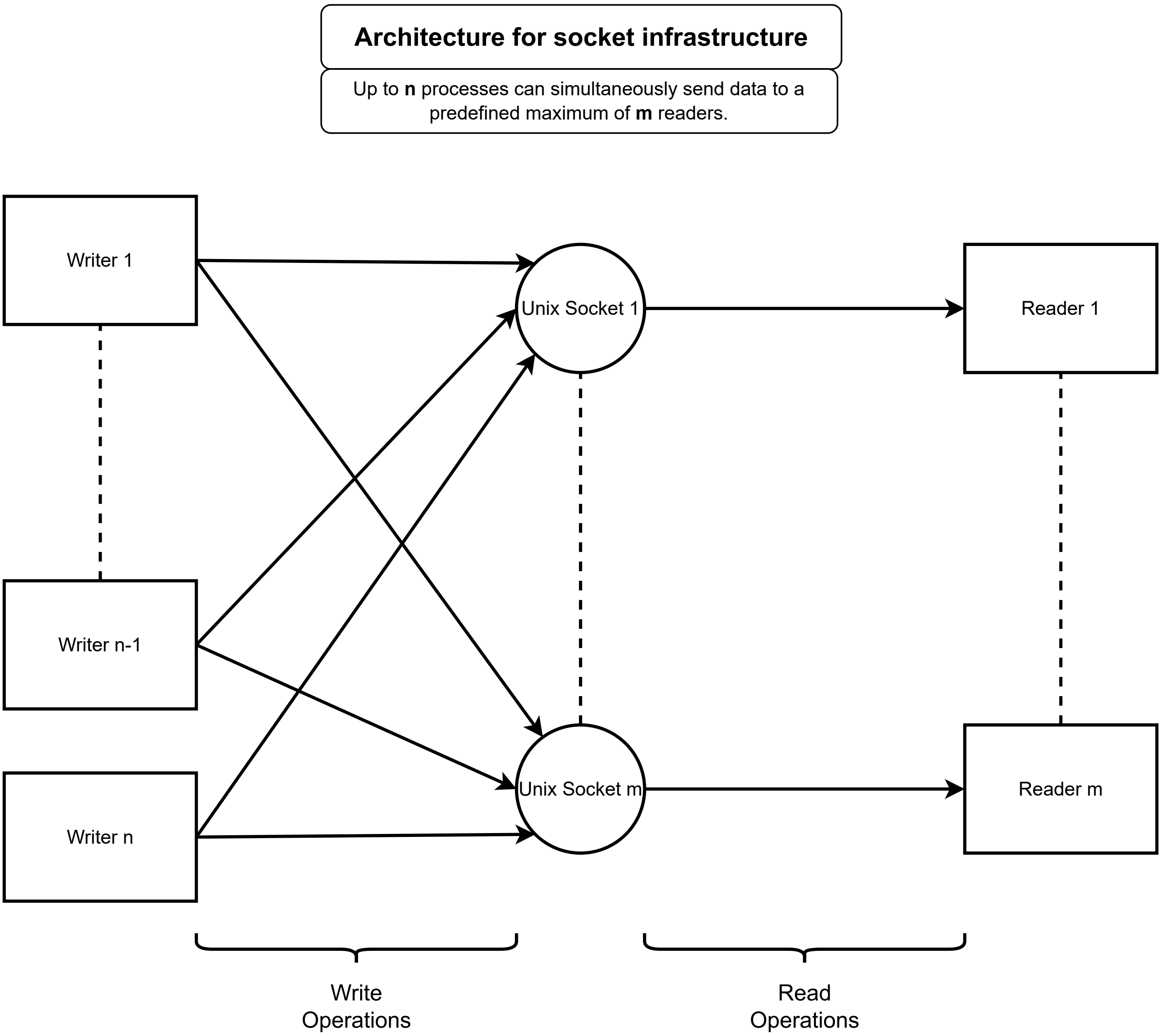

Architecture for socket infrastructure
Up to **n** processes can simultaneously send data to a predefined maximum of **m** readers.
Writer 1
Writer n-1
Writer n
Unix Socket 1
Unix Socket m
Reader 1
Reader m
Write Operations
Read Operations

Figure 3.1: Architecture for a n-reader and m-writers scenario using UNIX domain sockets.

4 Implementation

In this chapter, the implementation of the previously discussed design (Reference: ??) for the unix domain socket architecture in the programming language C is presented. This includes an explanation of the write and read API. All error codes are numeric and defined in the file `io_ipc.h`.

4.1 Auxiliary functions and structures

When utilizing the socket IPC type, some shared resources seen in ?? need to be set up. None of these are modified during runtime.

```
#define MAX_AMOUNT_OF_SOCKETS 32

// This has to be long enough to fit the number or the socket
    and a terminating null byte
#define SOCKET_TEMPLATE_LENGTH 128
#define SOCKET_NAME_TEMPLATE "/tmp/unixDomainSock4SF2B_"
```

Algorithm 4.1: Parameters shared between readers and writers.

To ensure that no writer gets stuck continually checking for new sockets, the global variable `MAX_AMOUNT_OF_SOCKETS` is defined. A special feature of the socket IPC type is the possibility of attaching a variable number of reader and writer processes, even during runtime. In fact, there is no actual limit for attaching new writer processes. Meanwhile, only up to `MAX_AMOUNT_OF_SOCKETS` reader processes can exists because of the strict one-to-one mapping between readers and sockets.

All unix domain sockets were bound to the filesystem, resulting in a common path to the location of all sockets needing to be supplied to both readers and writers. However, this `SOCKET_NAME_TEMPLATE` is not the full path to each socket. During runtime, each reader process trying to attach will append this name template with their own reader ID. The reader ID is determined by claiming the first ID not already in use. Since the length of the reader ID being appended to the `SOCKET_NAME_TEMPLATE` can vary, a maximum length for this template is defined in `SOCKET_TEMPLATE_LENGTH`.

Separating functions utilized by both readers and writers results in an unwieldy API. Shared usage of functions by both sides is achieved by supplying function calls with the role of the calling process, either `SOCK_WRITER` or `SOCK_READER`.

Therefore the function initializing communication between processes, `sock_init` displayed in ?? only requires a structure of parameters and the role of the calling process.

```
int sock_init(union sock_arg_t *sock_args,
              int role);
```

Algorithm 4.2: Initialization function for both reader and writer processes.

Defining a union containing both writer and reader structures as seen in ?? allows the user of the API to provide either one as a parameter for the same function. The actual purpose of `sock_init` is to enable connection between writer and readers by initiating the associated structure passed in the parameter `sock_args`. An explanation of both the `sock_writer_arg_t` and `sock_reader_arg_t` will follow in the next sections ?? and ??. Writers are provided with a list of possible locations of unix domain sockets belonging to reader processes. Meanwhile, readers are assigned a path, in which they create a unix domain socket. All sockets are set to be of the type SOCK_SEQPACKET.

```
union sock_arg_t{
    struct sock_writer_arg_t wargs;
    struct sock_reader_arg_t rargs;
};
```

Algorithm 4.3: Union containing either the parameters of a writer or reader process.

While other IPC types such as shared memory required an orderly detachment of writers and readers, this is not necessary for the socket approach. Instead, when terminating a reader process, only closure of the corresponding unix domain socket is necessary. Stopping a writer process currently results in deconstructing the entire unix domain socket architecture. This results in the functions `socket_finalize` and `socket_cleanup`, as shown in ?? and ?? respectively, being identical in behavior. In fact, `socket_finalize` simply calls `socket_cleanup` and was only provided in the socket API to make a seamless replacement of other finalize-style functions when switching IPC types possible.

```
int sock_finalize(union sock_arg_t *sock_args, int role);
```

Algorithm 4.4: Initializes cleanup of socket IPC.

```
int sock_cleanup(union sock_arg_t *sock_args, int role);
```

Algorithm 4.5: Cleanup of socket IPC.

4.2 Write API

The write API consists out of a single, versatile function: `sock_writev`. See ?? for its definition.

It requires four arguments:

- A pointer to an instance of the structure sock_writer_arg_t which will be introduced shortly.
- A pointer to an array of iovec structures. Each iovec structure defines separate memory regions of a variable size, acting as a buffer. An entire array of such structures represent a vector of memory regions[12].
- The integer invalid_count represents the number of log messages located in the iovec array.
- Finally, the maximum number of receiving sockets is given via the parameter maxNumOfSocks.

```
int sock_writev(struct sock_writer_arg_t *sock_args,
                struct iovec *log_iovs,
                uint16_t invalid_count,
```

Algorithm 4.6: Write API for the unix domain socket architecture

The structure sock_writer_arg_t contains all information regarding the sockets needed by the writer process, as seen in ??.

```
struct sock_writer_arg_t
{
    char socketPathNames[MAX_AMOUNT_OF_SOCKETS][
       SOCKET_TEMPLATE_LENGTH];
    struct sockaddr_un socketConnections[
       SOCKET_TEMPLATE_LENGTH];
    int socketRecvs[MAX_AMOUNT_OF_SOCKETS];
    int writeSockets[MAX_AMOUNT_OF_SOCKETS];
};
```

.

Algorithm 4.7: Writer structure containing critical information being reused over several calls of sock_writev

The first parameter socketPathNames is an array containing all possible paths in which unix domain sockets could be located. Here, the necessity for defining the variables MAX_AMOUNT_OF_SOCKETS and SOCKET_TEMPLATE_LENGTH becomes obvious. The entirety of the socket IPC is implemented as a static library. Therefore arrays can not be assigned a variable length during runtime. Another array, socketConnections, contains a collection of sockaddr_un structures. Each of these represents a single unix domain socket and their address information. The array socketRecvs stores integers displaying which sockets have already been connected to. Sockets can either be marked as not available (-1), available but not connected yet (0), or available and connected with the writer process (1). Lastly, clientSockets holds the file descriptor referring to each connected socket, as created by the function socket.

When calling sock_writev, the first thing being performed is a check for available unix domain sockets. This can be considered analogous to checking for new readers because of the strict one-to-one mapping between sockets and readers. If new sockets were identified, a connection with that socket is established and saved for future calls of this function. Then, all sockets with an existing connection are sent the data located in the iovec structures using the blocking function write. On success, the function returns the number of sent messages via the socket IPC.

4.3 Read API

The function sock_readv is responsible for reading data out of the unix domain socket infrastructure. As seen in ??, the function takes two arguments.

```
int sock_readv(struct sock_reader_arg_t *sock_args,
               struct iovec *iovecs);
```

Algorithm 4.8: Read API for the unix domain socket architecture

The parameter iovecs is a pointer to an array of iovec structures. Any log messages received via the socket IPC are stored here for the calling reader process to access later. A structure containing all relevant information regarding the specific unix domain socket associated with the reader process are stored in sock_args. This structure, sock_reader_arg_t, is defined in ??.

```
struct sock_reader_arg_t
{
        char socketPathName[SOCKET_TEMPLATE_LENGTH];
    struct sockaddr_un address;
        int sizeOfAddressStruct;
    int readSocket;
        int clientSockets[MAX_AMOUNT_OF_SOCKETS];
};
```

.

Algorithm 4.9: Reader structure containing critical information being reused over several calls of sock_readv

Analogous to sock_writer_arg_t, the path of the socket assigned to the reader process is passed in socketPathName. Parameter address contains the structure sockaddr_un representing that same unix domain socket. Not wanting to redetermine the static size of the address each function call, sizeOfAddressStruct is passed along containing that exact value. The integer readSocket contains the file descriptor referring to the readers unix domain socket, as created by the function socket. Saving already established connections with writer processes for future function calls is done in clientSockets.

Calling sock_readv creates a list of clients for the blocking function select will regularly poll.

select waits for at least one file descriptor (read: connection with writer process) to become ready for an I/O operation. A file descriptor is considered ready once a call of read or write will not block if performed.[13]

This stops the function sock_readv having to either be stuck in a blocking call of read, or return from a non-blocking call of read with an error code. Having a blocking call of select instead of read is desireable because it allows sock_readv to accept connections of new writer processes while waiting for data to arrive. Once select returns, sock_readv checks if new connections need to be accepted. If not, one of the already connected writer processes has sent data via the unix domain socket which is ready to be read. All received data is then saved in the provided parameter iovecs, allowing the calling process of sock_readv to access it. The function sock_readv will then terminate and return the number of received messages.

5 Experiments

Paul Raatschen has performed a study[5] concluding that the original Fail2ban process can be improved upon. It was determined that especially with many clients, Fail2ban struggled to keep up with high incoming traffic rates. To remedy this issue, a more performant program, Simplefail2ban, was implemented and measured. An increase in performance was evident. Simplefail2ban supported two modes of IPC (TODO: Abbreviation). The disk mode was akin to traditional file logging, while the shared memory approach would use a shared memory section to exchange data between processes. A direct comparison between the already outlined socket approach and previously supported IPC (TODO: Abbreviation) types necessitates the measurements in this chapter.

The following chapter details the conducted measurements, outlining specifics according to Jain's "The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling, NY: Wiley"[14] chapter 2.2.

5.1 Test environment

Two machines, both identical in hardware and software, were used in these experiments. The first machine, the device under test (DUT) (TODO: Abbreviation), ran Simplefail2ban and a test application responsible for receiving incoming traffic and reporting clients. The second machine generated and sent traffic, consisting of both valid and invalid traffic, to the DUT (TODO: Abbreviation) using TRex.

5.2 Experimental design

In his thesis[5] Paul Raatschen showed that the shared memory mode of Simplefail2ban outperforms the traditional Fail2ban. However, it remains unclear if the implementation of this IPC type is more performant than other alternatives. Specifically, the possibility of using Unix domain sockets as a mode of inter-process communication was not explored. The following experiments enable a direct comparison between the two IPC types.

In general, the experiments consist of two participants and a one-sided data exchange. The device under test (DUT) (TODO: Abbreviation), or more specifically the application udp_server, receives a stream of both wanted and unwanted data. Identifying desired traffic is done by analyzing the message payload. This is a crude and unrealistic approach to filtering malicious communication requests. Such a simplification allows the application udp_server to quickly generate log messages. Since the goal of this study is to determine the most efficient IPC type for Simplefail2ban, this abstraction does not diminish the findings of this thesis.

To compare the differing IPC (TODO: Abbreviation) types, a set of performance metrics needs to be established:

Performance metrics

- Total number of unwanted requests dropped (number of packets)
- Total number of unwanted requests dropped, relative to the total amount of unwanted requests send (percentage)
- Number of log messages processed by Simplefail2ban, relative to the number of log messages sent by the test server (percentage)
- Central Processing Unit (CPU) utilization of Simplefail2ban (seconds of CPU time)

Higher is better for the first three metrics. The last metric should be minimized for the DUT (TODO: Abbreviation) so its services are continually provided to valid clients.

The fixed parameters for each of the experiments are the following:

Fixed parameters

- Hardware and Software parameters of the testbed (TODO: Abbreviation) in this table:
  - CPU: 16 cores, no hyper-threading enabled
  - Network Interface Card (NIC): Maximum transfer unit, 1500 bytes
  - TRex: One interface, 30 threads

Table 5.1: Table of Hardware and Software parameters of the testbed. Both machines are identical. The first machine serves as the DUT, the second machine generates traffic to be sent to the DUT (TODO: Abbreviation, also for the table below) via TRex.

Hardware	
CPU	16 × Intel(R) Xeon(R) Silver 4314 CPU @ 2.40GHz
NIC	Mellanox Technologies MT2892 Family [ConnectX-6 Dx]
RAM	128 GB
Software	
OS	Debian GNU/Linux 11
Kernel	5.10.0-28-amd64 x86_64
NIC Driver	mlx5_core; Version 5.8-2.0.3
TRex	2.99 (Stateless)

- Number of entries in eBPF maps for IPv4 & IPv6: 1,000,000
- Number of receiving threads used by udp_server: 16
- Duration of measurement: 300 Seconds
- Amount of valid traffic sent : 50,000 PPS
- Number of clients sending valid traffic: 254
- **Simplefail2ban** parameters:
  - Number of hash table bins used: 6,000,011
  - Ban threshold for clients: 3
  - Ban time for clients: 30 seconds
  - Enabling the Regex Matching feature of Simplefail2ban (the current implementation does not ban clients correctly when disabled)
  - For **shared memory** specifically:
    - Number of banning threads used: 16
    - Line count for the shared memory buffer segments: 1,000,000
    - Segment count for the shared memory buffer: 16
    - Overwrite feature enabled
    - Workload stealing feature disabled
  - For **sockets** specifically:
    - Number of banning threads used: 16
    - Number of sockets: Same as number of reader processes
    - Using default path to sockets created by the application: tmp/
    - Using default socket receive and send buffer size configured on the system: 212992 Bytes
  - For **disk** specifically:
    - Number of banning threads used: 1 (disk mode only supports one banning threads)
    - Buffer size for uring_getlines: 2048

The factors, or variable parameters, during these experiments were the following:

Factors and their levels

- Effects of differing amount of invalid traffic sent: 100k, 1m, 10m, 20m, 30m PPS
- Effects of differing number of clients sending invalid data: 65,534 and 131,068

  - Range used for 65,534 clients: 10.4.0.1 to 10.4.255.254 resulting in clients stemming from 256 subnets (using offset_fixup of 5 for IPv6 in TRex script).
  - Range used for 131,068 clients: 10.4.0.1 to 10.5.255.252 resulting in clients stemming from 512 subnets (using offset_fixup of 5 for IPv6 in TRex script).
  - When using the IPv4/IPv6 IP stack, the range for 65,534 client is being used twice to generate both a IPv4 and IPv6 stream.

- IP stack: IPv4, IPv6 and IPv4/IPv6 mixed
- Differing IPC type: DISK, SHM, SOCK
- For shared memory specifically:
  - No 2nd Reader/ Enabling 2nd Reader
- For sockets specifically:
  - No 2nd Reader/ Enabling 2nd Reader

To generate the traffic being sent to the DUT, TRex scripts are used. These scripts provide the option to modify the sent traffic according to the factors outlined above. TODO: Add path in repo for these scripts During these measurements, adapted versions of Paul Raatschens[5] scripts were used. To measure most performance metrics, an adaptation of the xdp_ddos01_blacklist_cmdline program was used. This application originally stems from Florian Mikolajczak master's thesis[4] and routinely polls the number of dropped and passed packets from a specific eBPF map. It was modified by Paul Raatschen to output values as a csv file. The polled eBPF map is ultimately used by Simplefail2ban to ban clients. CPU time was measured via the command top.

5.3 Replicative experiments

Software version changes warrant remeasurement of the shared memory and disk IPC (TODO: Abbreviation) mode for Simplefail2ban. These will also be used to evaluate the newly implemented socket mode.

5.3.1 Experiment 1a: Replication of Simplefail2ban Logfile

It has already been shown that the disk mode of Simplefail2ban is outperformed by the shared memory mode. Pure IPv4, IPv6 and a mixed IPv4/IPv6 IP stack will be used. File logging is expected to perform worse than the other IPC (TODO: Abbreviation) modes discussed in this thesis. In total, 25 unique measurements were conducted for this experiment.

5.3.2 Experiment 1b: Replication of Simplefail2ban Shared Memory

The newly implemented socket approach is intended to be a valid alternative to the shared memory mode of Simplefail2ban. To enable a direct comparison, measurements for the shared memory mode need to be done under high loads, since with lower loads both the socket and shared memory mode are suspected to be performant enough. All levels of invalid traffic rates are measured individually. Again, either a pure IPv4, IPv6 or mixed IPv4/IPv6 IP stack is utilized. The most performant features will be used, meaning overwrite is enabled and workload stealing is disabled. No second reader process is being employed here. In total, 25 unique measurements were conducted for this experiment

5.4 Measuring the socket API

In the following section thorough variations of factors and their levels are used to conclude if the socket mode is reliable. Also, heavy workloads are employed to determine how the socket mode performs in worst case scenarios. This will enable a direct comparison between socket and shared memory mode. The data flow in the DUT can be seen in **??**.

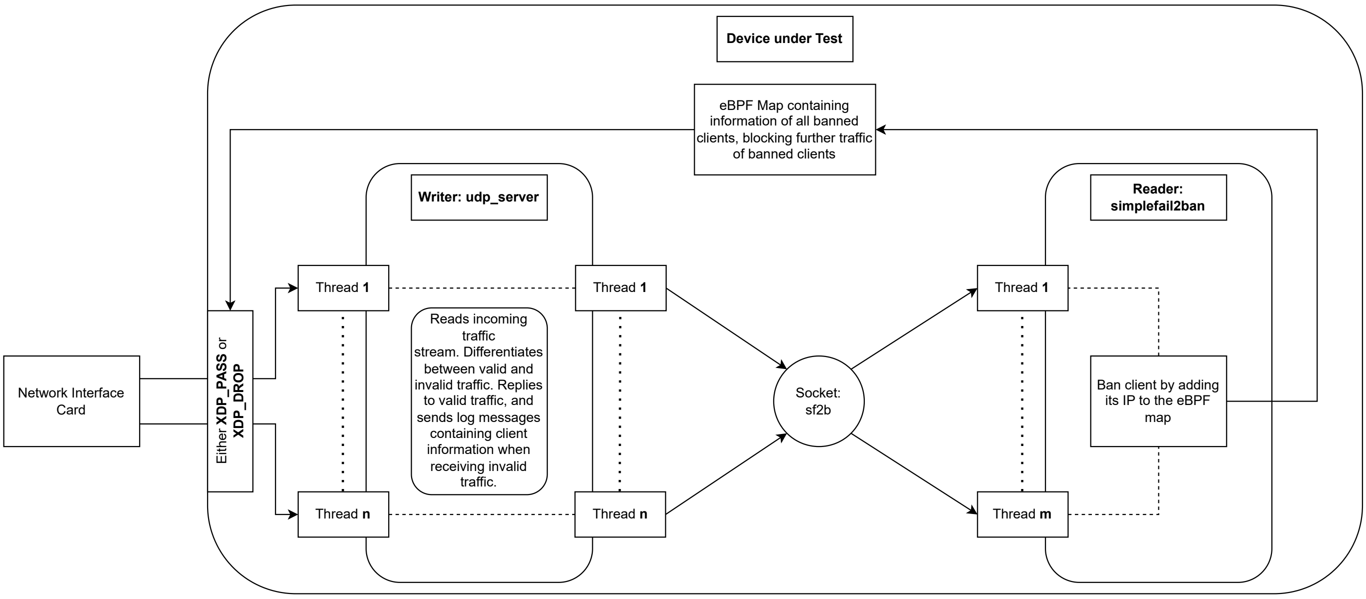

Device under Test
eBPF Map containing information of all banned clients, blocking further traffic of banned clients
Writer: udp_server
Reader: simplefail2ban
Network Interface Card
Either XDP_PASS or XDP_DROP
Thread 1
Thread n
Thread 1
Thread n
Reads incoming traffic stream. Differentiates between valid and invalid traffic. Replies to valid traffic, and sends log messages containing client information when receiving invalid traffic.
Socket: sf2b
Thread 1
Thread m
Ban client by adding its IP to the eBPF map

Figure 5.1: The graphic displays the data flow (left to right) on the DUT when enabling sockets as the IPC type. A packet can either be passed (XDP_PASS) or to the kernel or dropped (XDP_DROP) before ever reaching it.

5.4.1 Experiment 2: Simplefail2ban Sockets

To establish a baseline for the performance of the socket mode all factors are set to all possible levels in every combination. The only exception being the possibility of using a second reader process, which will have its own section later. In total, 25 unique measurements were conducted for this experiment.

5.4.2 Experiment 3a: Replication of Simplefail2ban Shared Memory with 2nd Reader

In order to later compare the socket mode and its option to have a second reader, a baseline measurement needs to be established. This experiment will be performed with 131,068 clients sending invalid data only and no pure IPv4 or IPv6 IP stack. Again, the overwrite feature is enabled while workload stealing is disabled. In total, 5 unique measurements were conducted for this experiment.

5.4.3 Experiment 3b: Simplefail2ban Sockets with 2nd Reader

This experiment closely mirrors the experiment 3a. A total of 131,068 clients will send invalid data to the DUT with no pure IPv4 or IPv6 IP stack. The shared memory mode inherently supports the possibility of adding a second reader to the shared memory section to read log messages. There is no such inherent support in the socket mode. Instead in its current implementation, another read process can be started which will then be assigned its own socket. This socket will then also receive all log messages. Consequently, the shared memory mode will likely see a larger performance gain, since no additional effort is required to send messages to the second reader. In total, 5 unique measurements were conducted for this experiment.

5.5 Evaluation of Experiments

The aforementioned experiments can be logically grouped in two categories: Baseline measurements and utilization of a second reader. Baseline measurements consist out of the experiments 1a, 1b and 2. TODO: Refer via link Meanwhile, the second reader experiments consist out of 3a and 3b. TODO: Refer via link With 85 performed experiments, a thorough yet not unreasonably long evaluation of each measurements is impossible. Instead, only especially expressive data will be covered in this section, with any notable or diverging observations being explicitly mentioned. For any readers interested in the data omitted from this thesis, or the repeat measurements performed to detect variations between each measurement, please refer to the repository provided in the sources[15].

Meaning of data variables

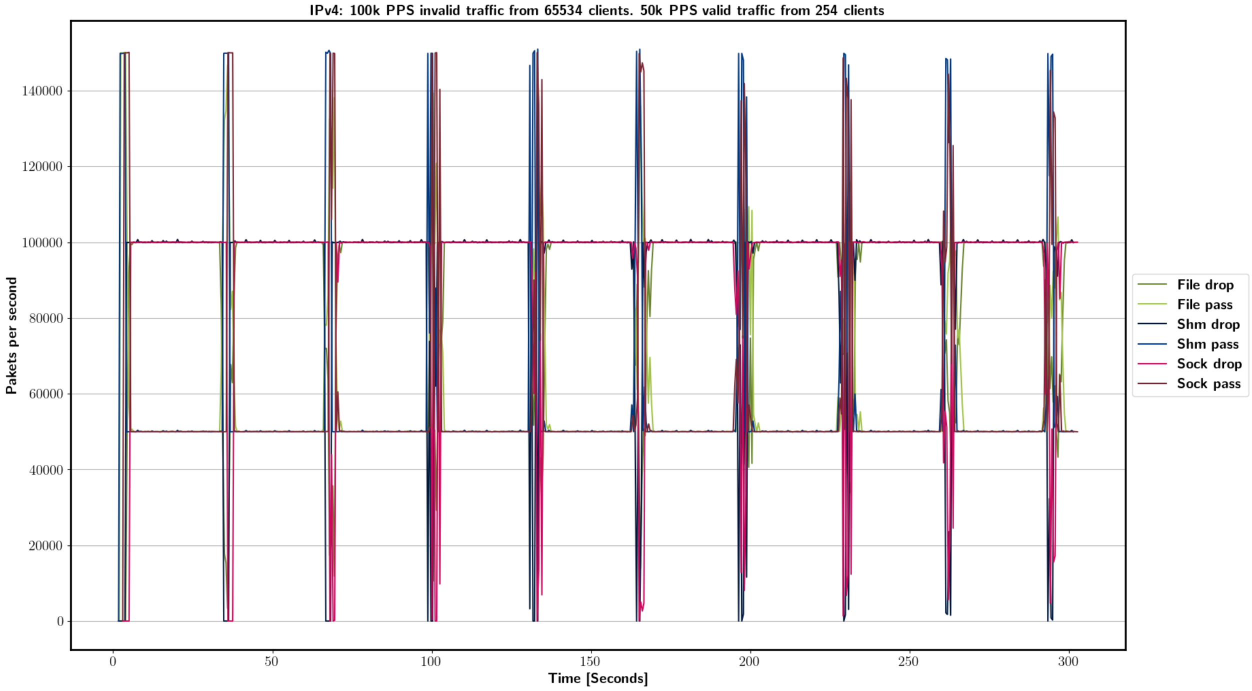
In the following section each graph will be accompanied with an additional table. This table contains data that is not explicitly expressed otherwise. A total of six lines are plotted, with two of them belonging to each IPC type. For each IPC type (File, Shm, Sock) the total number of packets dropped by the eBPF program is denominated via `XDP_DROP`. Similarly, the number of packets passed to the kernel is displayed via `XDP_PASS`. The `relative drop` represents the percentage of packets dropped relative to the theoretical maximal of dropped packets. Calculating the relative drop is done with the following formula: total dropped packets /(experiment duration $*$ invalid traffic rate $-$ number of ban cycles $*$ ban limit $*$ number of malicious clients) `Log messages` represents the number or messages sent via the chosen IPC type.

5.5.1 Baseline measurements

For this section, data of 75 experiments have been analyzed.

A trend displayed in **??** remains prevalent with lower rates of invalid traffic and especially when only 65534 clients send invalid data: Differences in performance are difficult to spot when graphed. The table in **??** does provide more information. All IPC types perform similarly well in defending the DoS attack. However, two anomalies stand out. The `relative drop` of the shared memory IPC type is over 100%. This happened a total of 21 times over separate measurements, but only when measuring traffic rates of 100k PPS. Why or how this is happening is unclear. It is suspected that the banning and unbanning threads of Simplefail2ban have intrinsic race conditions when trying to reban clients. The other notable thing is a stark difference in CPU time, with a definitive spike when using the socket IPC type. Otherwise, the results are as expected with the file IPC type performing worst.

This trend somewhat continues in **??**. Again, the graph plotting the number of dropped and passed packets is not definitive. Instead, a clear difference in performance is once again mainly visible in CPU time. The `relative drop` reveals that the shared memory IPC is performing best, being also supported by the lower number of packets passed to the kernel. An overall drop in performance measured via `relative drop` along all IPC types was expected. A generous best case scenario consists out of assuming that all 65534 clients are banned in the same timespan at all incoming traffic rates. But even then, the inherit latency (how ever miniscule it might be) of the IPS means that an increase in invalid



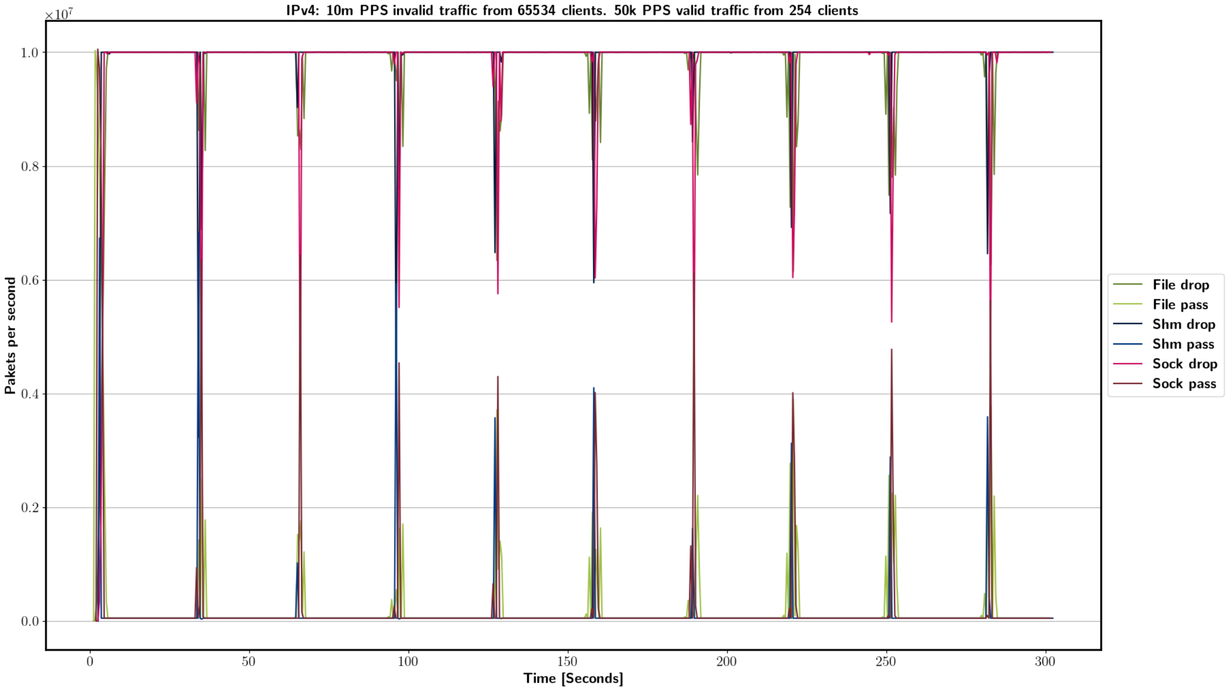
IPC type	XDP_DROP [10^6]	XDP_PASS [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
File	27,84	17,16	99,29900071	16,81	04.95
Shm	28,03	16,97	100,0000036	16,97	05.94
Sock	28,02	16,98	99,93706566	16,96	57.90

Figure 5.2: Total packets sent: 45m. Best case drop rate: 93,4466%

traffic flow directly correlates in more packets reaching the kernel during each ban cycle. Therefore, the `relative drop` rate must inversely correlate with the invalid traffic rate.

Even with 30m invalid PPS, as seen in **??**, all IPC types successfully defend against the DoS attack. The fact that the file IPC type outperforms the socket IPC type in terms of `relative drop` rate is especially noteworthy. File mode also outperforms shared memory and socket mode on CPU usage, which was not expected. With the high rate of incoming traffic during a ban cycle, the application udp_server had to wait on the IPC to be able to submit more log messages to Simplefail2ban. This resulted the system being unable to submit a substantial number of packets to udp_server. While not explicitly documented in this thesis, these number are available in the repository[15] of this thesis. The difference in log messages created by udp_server and number of passed packets correlates exactly with this observation. Measuring the number of packets unable to be submitted to udp_server was done by checking the file `/proc/net/udp6` on the DUT.

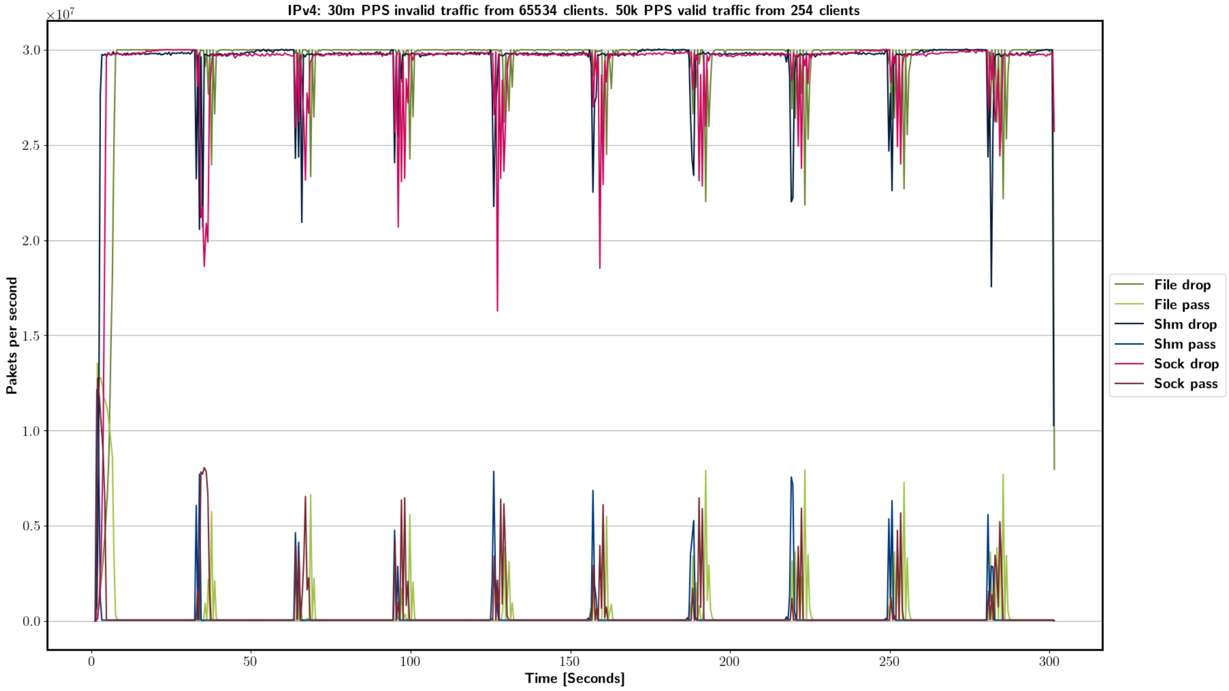
A mutual feature that these three measurements have is the inverse correlation between `relative drop` rate and number of sent `log messages`. Better performing IPC types



IPC type	XDP_DROP [10^8]	XDP_PASS [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
File	29,40	75,07	98,05973977	17,51	09.69
Shm	29,75	37,49	99,21848407	20,02	16.86
Sock	29,50	61,81	98,39458207	17,13	76.00

Figure 5.3: Total packets sent: 3015m. Best case drop rate: 99,934466%

are also able to send more log messages despite the fact that they block invalid traffic at an increased rate. This is explainable through the inability of the system to supply new packets to udp_server while it is still waiting on the IPC architecture to deliver data to the IPS. IPC types with lower latency and higher bandwidth can log more messages during the short influx of packets each ban cycle.



IPC type	XDP_DROP [10^8]	XDP_PASS [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
File	87,75	159,82	97,52375345	17,48	16.55
Shm	88,30	87,23	98,13105047	21,39	39.08
Sock	87,45	139,42	97,18179422	16,92	138.85

Figure 5.4: Total packets sent: 9015m. Best case drop rate: 99,97815533%

6 Conclusion & Outlook

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Erat nam at lectus urna. Vitae purus faucibus ornare suspendisse sed nisi lacus. Turpis egestas integer eget aliquet nibh praesent tristique magna. Et netus et malesuada fames ac turpis egestas. Nunc vel risus commodo viverra maecenas accumsan lacus. Nisi scelerisque eu ultrices vitae auctor eu augue. Odio morbi quis commodo odio aenean sed adipiscing. Ultricies lacus sed turpis tincidunt id aliquet. Sit amet mattis vulputate enim nulla aliquet porttitor lacus luctus. Tellus rutrum tellus pellentesque eu tincidunt tortor.

6.1 Evaluation of socket API

Evaluation of socket API

List of Figures

List of Tables

List of Algorithms

A Abbreviations

B Source Files

The source files and the corresponding repository can be accessed by contacting the second supervisor: Max Schrötter.

Bibliography

[1] James P. Anderson. “Computer Security Threat Monitoring and Surveillance”. In: *James P. Anderson Company* (1980).

[2] Dorothy E. Denning. “An Intrusion-Detection Model”. In: *IEEE Transactions on Software Engineering* SE-13.2 (1987).

[3] Letou Kopelo, Devi Dhruwajita, and Y. Jayanta Singh. “Host-based Intrusion Detection and Prevention System (HIDPS)”. In: *International Journal of Computer Applications* 69.(0975 - 8887) (2013).

[4] Florian Mikolajczak. “Implementation and Evaluation of an Intrusion Prevention System Leveraging eBPF on the Basis of Fail2Ban”. MA thesis. University of Potsdam, 2022.

[5] Paul Raatschen. *Design and Implementation of a new Inter-Process Communication Architecture for Log-based HIDS for 100 GbE Environments*. Bachelorthesis. 2023.

[6] Linux man-pages project. *Socket*. https://man7.org/linux/man-pages/man2/socket.2.html. Last visited on 09-07-2024. 2024.

[7] Brian "Beej Jorgensen" Hall. *Beej’s Guide to Network Programming: Using Internet Sockets*. 2023.

[8] Linux man-pages project. *Unix Domain Socket*. https://man7.org/linux/man-pages/man7/unix.7.html. Last visited on 09-07-2024. 2024.

[9] Cisco Systems. *Official TRex Website*. https://trex-tgn.cisco.com/. Last visited on 09-07-2024.

[10] The Linux Foundation. *Official DPDK Website*. https://www.dpdk.org/. Last visited on 09-07-2024.

[11] IEEE Computer Society and The Open Group. *IEEE P1003.1™, Draft 3*. Tech. rep. 2007.

[12] Linux man-pages project. *Iovec*. https://man7.org/linux/man-pages/man3/iovec.3type.html. Last visited on 11-07-2024. 2024.

[13] Linux man-pages project. *Select*. https://man7.org/linux/man-pages/man2/select.2.html. Last visited on 12-07-2024. 2024.

[14] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques For Experimental Design, Measurement, Simulation, and Modeling, NY: Wiley*. John Wiley & Sons, Inc., 1991.

[15] Daniel Aeneas von Rauchhaupt. *Thesis Git Repository*. https://gitup.uni-potsdam.de/fips/fips_sock. Repository containing all source code and measurement data. 2024.

[16] *Fail2ban GitHub*. https://github.com/fail2ban/fail2ban/wiki/How-fail2ban-works. 2017.